FreeST: Context-free Session Types in a Functional Language

Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos

ETAPS PLACES 2019

Abstract

FreeST is an experimental concurrent programming language. Based on a core linear functional programming language, FreeST features primitives to fork new threads, to create channels, and to communicate on these. A powerful type system of context-free session types governs the interaction on channels. The compiler builds on a novel algorithm for deciding type equivalence of context-free session types.

Kinds

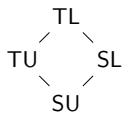
FreeST requires kinding. And the reason is polymorphism, not context-free types.

- !**Int**; ?**Bool** session type
- **Int** -> **Bool** functional type
- !**Int**;α **?**

!**Int**;α session type only if α is session type

Kind = [Prekind] [Multiplicity]

T functional **L** linear
S session **U** unrestricted



Types

The functional types are:

- Basic types: **Int**, **Bool**, **Char**, and ()
- Unrestricted functions: T_1 –> T_2
- Linear functions: T_1 –o T_2
- Pairs: (T_1, T_2)
- Datatypes: [l_1 : T_1, ..., l_n : T_n]

The session types are:

- Neutral: **Skip**
- Sequential composition: S_1; S_2
- Messages: !B and ?B
- Choices: + {l_1 : S_1, ..., l_n : S_n} and
- & {l_1 : S_1, ..., l_n : S_n}
- Recursive types: **rec** x . S

Expressions inspired in functional languages

FreeST blends expressions typical of functional languages and of session types. The expressions inspired from functional languages include:

- Basic values: ints, bools, chars, and ()
- Term variables
- Lambda introduction:
  \x -o e for linear abstractions
  \x -> e for unrestricted abstraction
- Lambda elimination, e_1 e_2
- Pair introduction, (e_1, e_2)
- Pair elimination, **let** x, y = e_1 **in** e_2
- Datatype elimination,
  case e of l_1 x_{11}...x_{1k} -> e_1, ..., l_n x_{n1}...x_{nk} -> e_n
- Conditional expressions **if** e_1 **then** e_2 **else** e_3
- Type application, x[T_1, ..., T_n]
- Thread creation, **fork** e

Expressions inspired in session types

The session-type related expressions are:

- Channel creation, **new** S
- Message **send** E and **receive** e
- Branch selection, **select** l e
- Branch match,
  match e **with** l_1 x –> e_1 ,..., l_n x –> e_n

Polymorphism

Polymorphic variables are introduced with the **forall** construct. The polymorphic type

transform : **forall** α => Tree –>
TreeC; α –> (Tree, α)

has different types for different calls to function transform.

Type Equivalence

sound and complete

The compiler embeds an algorithm to check type equivalence. Deciding the equivalence of types relies on the construction of a finite relation whose least congruence with respect to sequential composition coincides with the bisimulation. The algorithm has 3 main stages:

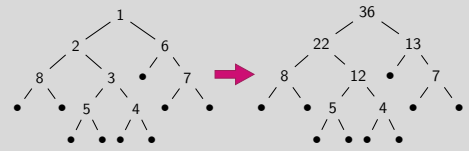
1. Convert types to a context-free grammar
Translates types into a set of productions

2. Prune unnormed productions
Streamlines the grammar by pruning unnormed productions

3. Simplify and expand
Alternates between simplification and expansion operations, until reaching a successful branch in the expansion tree or concluding that all branches are unsuccessful.

Example

Serialize a tree object on a channel. The aim is to transform a tree by interacting with a remote server. The client process streams a tree on a (single) channel. The server process reads a tree from the other end of the channel and, for each node received, sends back the sum of the integer values under (and including) that node.




```
data Tree = Leaf | Node Int Tree Tree
type TreeC = +{Leaf: Skip , Node: !Int; TreeC; TreeC; ?Int}
type TreeS = &{Leaf: Skip , Node: ?Int; TreeS; TreeS; !Int}

transform : forall α => Tree  –> TreeC; α –> (Tree, α)
transform tree c =
  case tree of {
    Leaf –>
      (Leaf , select Leaf c);                    -- c : Skip
    Node x l r –>
      let c = select Node c in                   -- c : !Int; TreeC; TreeC; ?Int; α
      let c = send x c in                        -- c : TreeC; TreeC; ?Int; α
      let l , c = transform[TreeC ; ?Int ; a] l c in -- c : TreeC;?Int;α
      let r , c = transform[?Int ; a] r c in     -- c : ?Int;α
      let y , c = receive c in                   -- c : α
      (Node y l r, c)
  }

treeSum : forall a => TreeS;a –> (Int, a)
treeSum c =
  match c with {
    Leaf c –>
      (0 , c);                                   -- c : Skip
    Node c –>
      let x , c = receive c in                   -- c : ?Int; TreeS; TreeS; !Int; α
      let l , c = treeSum[TreeS;!Int;a] c in     -- c : TreeS;?Int;α
      let r , c = treeSum[!Int;a] c in           -- c :?Int;α
      let c = send (x + l + r) c in              -- c : α
      (x + l + r, c)
  }

main: Tree
main =
  let w, r  = new  TreeC  in
  let _     = fork (treeSum[Skip] r) in
  let t, _  = transform[Skip] aTree w in
  t
```

```
-- Target Haskell code
{-# LANGUAGE BangPatterns #-}
import FreeSTRuntime

data Tree = Leaf | Node Int Tree Tree deriving Show

treeSum !c =
  _receive c >>= \(l, c) ->
  case l of
    "LeafC" ->
      return (0, c)
    "NodeC" ->
      _receive c >>=
      \(x, c) -> treeSum c >>=
      \(l, c) -> treeSum c >>=
      \(r, c) -> _send (x + l + r) c >>=
      \c -> return (x + l + r, c)

transform !tree !c = ...

main =
  _new >>=
  \(w, r) -> _fork (treeSum r >> return ()) >>
  transform aTree w >>=
  \(t, _) -> return t
```

```
module FreeSTRuntime (_fork, _new, _send, _receive) where
import Control.Concurrent (forkIO)
import Control.Concurrent.Chan.Synchronous (newChan, writeChan, readChan)
import Unsafe.Coerce (unsafeCoerce)

_fork e = forkIO e >> return ()
_new = newChan >>= \ch -> return (ch, ch)
_send x ch  = writeChan ch (unsafeCoerce x) >> return ch
_receive ch = readChan ch >>= \a -> return (unsafeCoerce a, ch)
```

FreeST is a polymorphic functional language with context-free session types

Features full type equivalence via a novel algorithm embedded in the compiler

FreeST generates Haskell code that can be later compiled with an Haskell compiler

Future:

- Type application inference
- Transmission of arbitrary types

Acknowledgments: This work was supported by FCT through project Confident (PTDC/EEICTP/4503/2014), by the LASIGE research Unit (UID/CEC/00408/2019) and by Cost Action CA15123 EUTypes, supported by COST (European Cooperation in Science and Technology).

U LISBOA | Ciências ULisboa | FCT Fundação para a Ciência e a Tecnologia | LASIGE driven by excellence